

Java 异常

异常是Java提供的一种识别及响应错误的一致性机制。从而可以达到程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性。

定义：异常就是有异于常态，和正常情况不一样，有错误出现。在java中，将程序执行过程中的不正常情况称之为异常，开发过程中的语法错误和逻辑错误不是异常，发生异常时java会阻止当前方法或作用域的情况。

异常：指的是程序在执行过程中，出现的非正常的情况，最终会导致JVM的非正常停止。

异常指的并不是语法错误,语法错了,编译不通过,不会产生字节码文件,根本不能运行。

在Java等面向对象的编程语言中，异常本身是一个类，产生异常就是创建异常对象并抛出了一个异常对象。

Java为异常设计了一套异常处理机制，当程序运行过程中发生一些异常情况时，程序不会返回任何值，而是抛出封装了错误信息的异常对象。这样保证程序代码更加优雅，并提高程序的健壮性。为什么要设计异常呢？首先，引入异常之后，我们就可以把错误的代码从正常代码中分离出来进行单独处理，这样使代码变得更加整洁；其次，当出现一些特殊情况时，我们还可以抛出一个检查异常，告知调用者让其处理。



Throwable

所有的异常都是从Throwable继承而来的，是所有所有错误与异常的超类。Throwable包含了其线程创建时线程执行堆栈的快照，它提供了printStackTrace()等接口用于获取堆栈跟踪数据等信息。

- 而Throwable体系下包含有两个子类，Error（错误）和Exception（异常），它们通常用于指示发生了异常情况。二者都是Java异常处理的重要子类，各自都包含大量子类。

Error（错误）

- **定义：**Error类及其子类。程序中无法处理的错误，表示运行应用程序中出现了严重的错误。大多数错误与代码编写者执行的操作无关，而是表示代码运行时JVM出现的问题。
- **特点：**对于所有的编译时期的错误以及系统错误都是通过Error抛出的。这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时。通常有如Virtual MachineError（虚拟机运行错误）等。当JVM不再

有继续执行操作所需的内存资源时，将出现 `OutOfMemoryError`（内存溢出错误），还有 `StackOverflowError`（栈溢出错误）等。这些异常发生时，JVM 一般会选择**线程终止**。

- **注意：**这些错误是不受检查的，非代码性错误，不可查的。因为它们在实际应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。因此，当此类错误发生时，应用程序不应该去处理此类错误。

Exception（异常）

Exception 是另外一个非常重要的异常子类。程序本身可以捕获并且可以处理异常。这类异常一旦出现，我们就要对代码进行更正，修复程序。Exception 这种异常又分为两类：运行时异常和编译时异常。

运行时异常

- **定义：**`RuntimeException` 类及其子类异常，如 `NullPointerException`（空指针异常）、`IndexOutOfBoundsException`（下标越界异常）等，表示 JVM 在运行期间可能出现的异常。
- **特点：**此类异常，Java 编译器不会检查它，属于不受检查的异常。一般是由程序逻辑错误引起的，此类程序应该从逻辑角度尽可能避免这类异常的发生。而当程序中可能出现这类异常，即使没有用 `try-catch` 语句捕获它，也没有用 `throws` 子句声明抛出它，也会编译通过。在程序中可以选 择捕获处理，也可以不处理。如果产生运行异常，则需要通过修改代码来进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！
- **注意：**`RuntimeException` 异常会由 JVM 自动抛出并自动捕获（**就算我们没写异常捕获语句运行时也会抛出错误！！**），此类异常的出现绝大多数情况是代码本身有问题，应该从逻辑上去解决并改进代码。这里我们来看下运行时异常是怎样的，这里我想说下，出现异常,不要紧张,把异常的简单类名,拷贝到 API 中去查。然后看是什么异常。可以看出，我们的程序逻辑出现错误，所以出现了算术异常。我们只要修改 `int b = 10` 就行

了，或者b不等于0都可以。所以遇到异常，我们不用担心。可以先从查看异常类名开始，看是什么异常，看是什么原因，找到我们程序出错的地方并进行修改就可以正常运行了。

- 那我们什么都没有处理，那出现异常时，是谁处理了这个异常呢？是JVM的默认处理：把异常的名称,原因,位置等信息输出在控制台，但是呢程序就不能继续执行了。

非运行时异常（编译时异常）

- **定义：**Exception中除 RuntimeException 及其子类之外的异常。
- **特点：**此类异常，Java 编译器会检查它。如果程序中出现此类异常，从程序语法角度讲是必须进行处理得异常。例如：
ClassNotFoundException（没有找到指定的类异常），IOException（IO流异常），要么通过throws 进行声明抛出，要么通过[try-catch](#)进行捕获处理，否则不能通过编译。
- **注意：**在程序中，通常我们不会自定义该类异常，而是直接使用系统提供的异常类。**该异常我们必须手动在代码里添加捕获语句来处理该异常。**通过注释可以看到，createNewFile() 方法是处理了IOException异常的，而IOException异常又继承来自Exception，是非运行时异常，所以必须处理异常。所以我们如果是编译时异常，在编译时期就报错了，必须处理这个异常，不然程序不能编译通过。

受检异常与非受检异常

通常，Java的异常（Throwable）分为**受检异常**（checked exceptions）和**非受检异常**（unchecked exceptions）。

受检异常

编译器要求必须处理得异常。

- 正确的程序在运行过程中，经常容易出现的、符合预期的异常情况。一旦发生此类异常，就必须采用某种方式进行处理。**除了Exception中的**

RuntimeException 及其子类以外，其他的 Exception 类及其子类异常就是非运行时期异常都属于受检异常。

- 这种异常编译器会检查它，也就是说当编译器检查到应用中的某处可能会此类异常时，将会提示你处理本异常——要么使用 try-catch 捕获，要么使用方法签名中用 throws 关键字抛出，否则编译不通过。

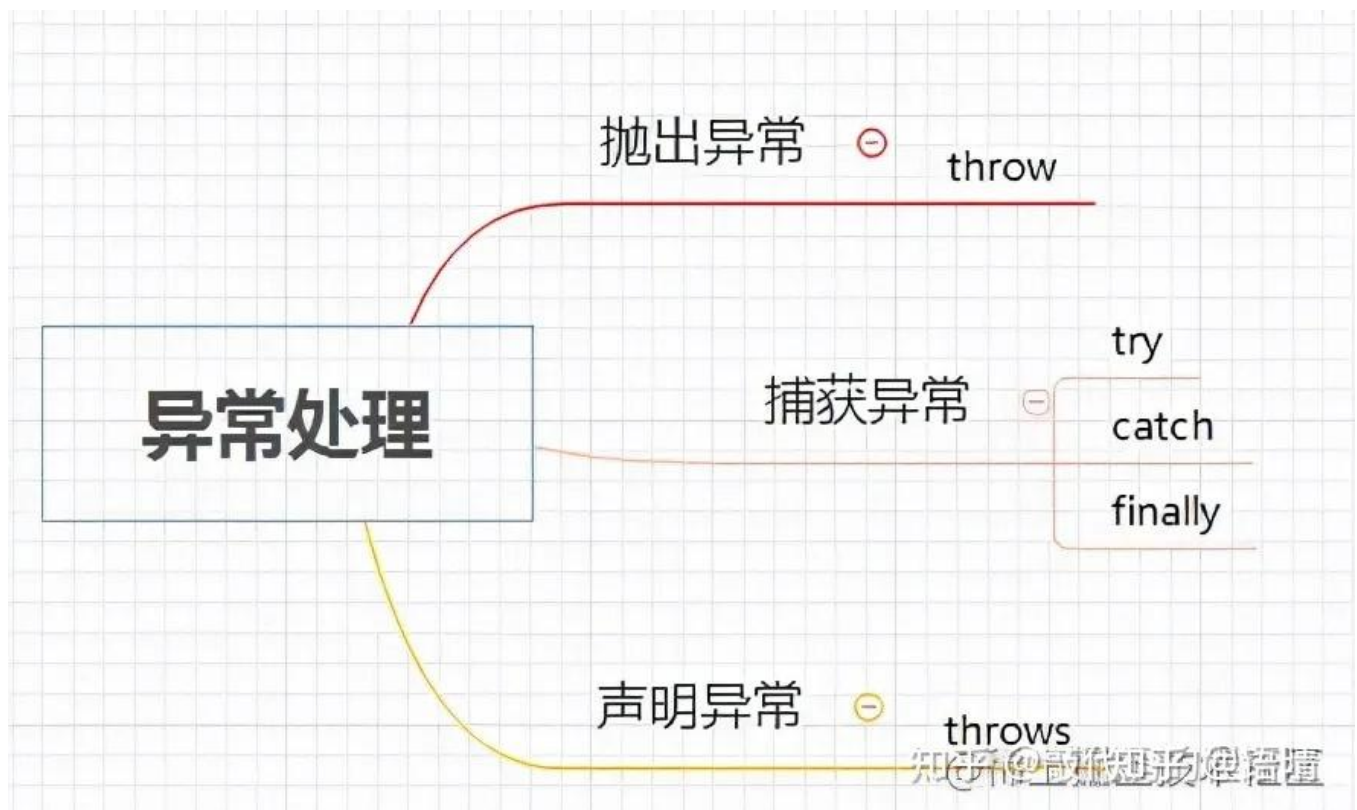
非受检异常

编译器不会进行检查并且不要求必须处理的异常。

- 此类异常，就是当程序中出现此类异常时，即使我们没有 try-catch 捕获它，也没有使用 throws 抛出该异常，编译也会正常通过。该类异常包括运行时异常（RuntimeException 及其子类）和错误（Error）。
RuntimeException 发生的时候，表示程序中出现了编程错误，所以应该找出错误修改程序，而不是去捕获 RuntimeException。

异常的处理机制

在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。



- 在Java中，一旦方法抛出异常，系统自动根据该异常对象寻找合适异常处理器（Exception Handler）来处理该异常，把各种不同的异常进行分类，并提供了良好的接口。
- 在 Java 中，每个异常都是一个对象，它是 Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个方法可以捕获到这个异常并可以对其进行处理。
- Java 的异常处理涉及了 5 个关键词：try、catch、finally、throw 和 throws。
- 在Java应用中，异常的处理机制分为声明异常throws，抛出异常throw和捕获异常try、catch、finally。接下来让我为大家详细讲述吧。

异常处理的关键词

- **throw**：用于抛出异常。
- **try**：用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。
- **catch**：用于捕获异常。catch用来捕获try语句块中发生的异常。
- **finally**：finally语句块总是会被执行。它主要用于回收在try块里打开的资源(如数据库连接、网络连接和磁盘文件)。注意：只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw 语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。
- **throws**：用在方法签名中，用于声明该方法可能抛出的异常。

这里先了解下关键词，具体定义格式和使用方法在下面介绍：

抛出异常throw

- 那什么时候使用呢？
1. 比如，在定义方法时，方法需要接受参数。那么，当调用方法使用接受到的参数时，首先需要先对参数数据进行合法的判断，数据若不合法，就应该告诉调用者，传递合法的数据进来。这时需要使用抛出异常的方式来告

诉调用者。

2. 或者当你觉得解决不了某些异常问题，且不需要调用者处理，那么你也可以抛出异常。

- **throw的作用**：在方法内部抛出一个Throwable 类型的异常。任何Java 代码都可以通过throw语句抛出异常。
- 具体如何抛出一个异常呢？

1. 创建一个异常对象。封装一些提示信息(信息可以自己编写)。
2. 需要将这个异常对象告知给调用者。怎么告知呢？怎么将这个异常对象传递到调用者处呢？通过关键字throw就可以完成。throw异常对象。
throw用在方法内，用来抛出一个异常对象，将这个异常对象传递到调用者处，并结束当前方法的执行。

- 定义格式：

- 例子：

```
throw new NullPointerException("要访问的arr数组不存在");  
throw new ArrayIndexOutOfBoundsException("该索引在数组中不存在，已超出范围");
```

接下来用程序具体演示下吧：

```
public class ThrowDemo {  
    public static void main(String[] args) {  
        //创建一个数组  
        int[] arr = {2,4,52,2};  
        //根据索引找对应的元素  
        int index = 4;  
        int element = getElement(arr, index);  
        System.out.println(element);  
        System.out.println("over");  
    }  
    /*
```

```
    * 根据 索引找到数组中对应的元素
    */
    public static int getElement(int[] arr,int index){
        //判断索引是否越界
        if(index<0 || index>arr.length-1){
            /*
                判断条件如果满足，当执行完throw抛出异常对象后，方法已经无法继续运算。
                这时就会结束当前方法的执行，并将异常告知给调用者。这时就需要通过异常来解决
            */
            throw new ArrayIndexOutOfBoundsException("你的索引越界了");
        }
        int element = arr[index];
        return element;
    }
}
```

运行后输出结果为：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 你的索引越
    at com.it.test2.ThrowDemo.getElement(ThrowDemo.java:25)
    at com.it.test2.ThrowDemo.main(ThrowDemo.java:10)
```

可以看到我定义了索引为4，但是数组的长度只有4。所以会报错。

注意：所以如果产生了问题，我们就会throw将问题描述类即异常进行抛出，也就是将问题返回给该方法的调用者。结果是ArrayIndexOutOfBoundsException的数组索引越界的问题。

那么对于调用者来说，该怎么处理呢？一种是进行捕获处理，另一种就是继续讲问题声明出去，使用throws声明处理。

声明异常throws

如果一个方法可能会出现异常，但没有能力处理这种异常，可以在方法声明处

用throws子句来声明抛出异常。例如汽车在运行时它可能会出现故障，汽车本身没办法处理这个故障，那就让开车的人来处理。

- **声明异常：**将问题标识出来，报告给调用者。如果方法内通过throw抛出了编译时异常，而没有捕获处理（稍后讲解该方式），那么必须通过throws进行声明，让调用者去处理。

关键字throws运用于方法声明之上,用于表示当前方法不处理异常,而是提醒该方法的调用者来处理异常(抛出异常)。

- **定义格式：** throws语句用在方法定义时声明该方法要抛出的异常类型，如果抛出的是Exception异常类型，则该方法被声明为抛出所有的异常。多个异常可使用逗号分割。

修饰符 返回值类型 方法名(参数) throws 异常类名1,异常类名2...{ }

注意：当方法抛出异常列表的异常时，方法将不对这些类型及其子类类型的异常作处理，而抛向调用该方法的方法，由他去处理。使用throws关键字将异常抛给调用者后，如果调用者不想处理该异常，可以继续向上抛出，但最终要有能够处理该异常的调用者。比如汽车坏了，开车的人也不会修理，只能叫修车公司来修理了。

- **演示一下：**一般来说，throws和 throw通常是成对出现的，例如：

```
public class ThrowsDemo {
    public static void main(String[] args) throws FileNotFoundException {
        readFile("a.txt");
    }
    // 如果定义功能时有问题发生需要报告给调用者。可以通过在方法上使用throws关键字进行声明
    public static void readFile(String path) throws FileNotFoundException {
        if(!path.equals("a.txt")) { //如果不是 a.txt这个文件
            // 我假设 如果不是 a.txt 认为 该文件不存在 是一个错误 也就是异常 throw
            throw new FileNotFoundException("文件不存在");
        }
    }
}
```

```
}  
}
```

而throws用于进行异常类的声明，若该方法可能有多种异常情况产生，那么在throws后面可以写多个异常类，用逗号隔开。

```
public class ThrowsDemo2 {  
    public static void main(String[] args) throws IOException {  
        readFile("a.txt");  
    }  
    //若该方法可能有多种异常情况产生，那么在throws后面可以写多个异常类，用逗号隔开  
    //若有异常a是异常b的子类，也可以直接省略，写b异常  
    private static void readFile(String path) throws FileNotFoundException, IOException {  
        if (!path.equals("a.txt")) { //如果不是 a.txt这个文件  
            // 我假设 如果不是 a.txt 认为 该文件不存在 是一个错误 也就是异常 throw  
            throw new FileNotFoundException("文件不存在");  
        }  
        if (!path.equals("b.txt")) {  
            throw new IOException();  
        }  
    }  
}
```

- throws抛出异常的规则：

1. 如果是非受检异常（unchecked exception），即Error、RuntimeException或它们的子类，那么可以不使用throws关键字来声明要抛出的异常，编译仍能顺利通过，但在运行时会被系统抛出。
2. 如果一个方法可能出现受检异常（checked exception），要么用try-catch语句捕获，要么用throws子句声明将它抛出，否则会导致编译错误。
3. 只有当抛出了异常时，该方法的调用者才必须处理或者重新抛出该异常。若当方法的调用者无力处理该异常的时候，应该继续抛出。

4. 调用方法必须遵循任何可查异常的处理和声明规则。若覆盖一个方法，则声明与覆盖方法不同的异常。声明的任何异常必须是被覆盖方法所声明异常的同类或子类。

捕获异常try、finally、catch

这三个关键字主要有下面几种组合方式try-catch、try-finally、try-catch-finally。

注意：catch语句可以有一个或者多个或者没有，finally至多有一个，try必要有。

那这里你会问有没有单独try模块出现，那我想问下你，try是用来监听是否有异常，那如果发生了异常，谁来捕获呢？所以没有try单独出现。而且编译不能通过。

所以跟try模块一样，例如catch，finally也不能单独使用出现

- 程序通常在运行之前不报错，但是运行后可能会出现某些未知的错误，不想异常出现导致程序终止，或者不想直接抛出到上一级，那么就需要通过try-catch等形式进行异常捕获，之后根据不同的异常情况进行相应的处理。
- **捕获异常：**Java中对异常有针对性的语句进行捕获，可以对出现的异常进行指定方式的处理。

捕获异常语法如下：

try-catch 形式：

```
try{  
    编写可能会出现异常的代码  
}catch(异常类型 e){  
    处理异常的代码  
    //记录日志/打印异常信息/继续抛出异常  
}
```

例如：

```
public class TryCatchDemo {
    public static void main(String[] args) {
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd")
        try {
            //当产生异常时，必须有处理方式。要么捕获，要么声明。
            Date date = simpleDateFormat.parse("2020-10-06");
        } catch (ParseException e) { // 括号中需要定义什么呢？
            //try中抛出的是什么异常，在括号中就定义什么异常类型
            e.printStackTrace(); //记录日志/打印异常信息/继续抛出异常
        }
        /*
            public Date parse(String source) throws ParseException{}
            //parse抛出了ParseException异常
            public class ParseException extends Exception {}
        */
    }
}
```

如何获取异常信息：

Throwable类中定义了一些查看方法：

- public String getMessage():获取异常的描述信息,原因(提示给用户的时候,就提示错误原因。
- public String toString():获取异常的类型和异常描述信息。
- public void printStackTrace():打印异常的跟踪栈信息并输出到控制台。
- 具体我们可以来看下：

```
public class TryCathDemo2 {
    public static void main(String[] args) {
```

```

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
try {
    //当产生异常时，必须有处理方式。要么捕获，要么声明。
    //演示下获取异常信息，修改了格式。
    Date date = simpleDateFormat.parse("2020年10月06");
} catch (ParseException e) {
    //public String getMessage():获取异常的描述信息,原因(提示给用户的时候,就提
    System.out.println(e.getMessage()); //Unparseable date: "2020年10月06"
    System.out.println(e.toString()); //java.text.ParseException: Unparse
    e.printStackTrace(); //输出信息而且飘红!!!
    /*
    java.text.ParseException: Unparseable date: "2020年10月06"
        at java.text.DateFormat.parse(DateFormat.java:366)
        at com.it.test3.TryCathDemo2.main(TryCathDemo2.java:13)
    */
}
}
}

```

而如果有多个异常使用捕获我们又该如何处理呢？

1. 多个异常分别处理。
2. 多个异常一次捕获，多次处理。

一般我们是使用一次捕获多次处理方式，格式如下：

```

try{
    编写可能会出现异常的代码
}catch(异常类型A e){ 当try中出现A类型异常,就用该catch来捕获.
    处理异常的代码
    //记录日志/打印异常信息/继续抛出异常
}catch(异常类型B e){ 当try中出现B类型异常,就用该catch来捕获.
    处理异常的代码
    //记录日志/打印异常信息/继续抛出异常
}
}

```

例如：

```
public class TryCatchDemo3 {
    public static void main(String[] args) {
        //test();
        test2();
    }

    //多个异常一次捕获，多次处理。
    public static void test2(){
        int[] arr = {11, 22, 66, 0};
        try {
            //System.out.println(arr[5]);//一旦这个报错，下面的代码就不会执行
            System.out.println(arr[2]);
            System.out.println(arr[0] / arr[arr.length-1]);
        } catch (ArithmeticException e) {
            System.out.println("除数不为0");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("数组下标越界");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //分别处理的方式
    public static void test() {
        int a = 10;
        int b = 0;

        try {
            System.out.println(a / b);
        } catch (ArithmeticException e) {
            System.out.println("除数不为0");//除数不为0
        }

        int[] arr = {1, 2, 3};
```

```
try {
    System.out.println(arr[4]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("数组下标越界");//数组下标越界
}
}
```

注意：一次捕获，多次处理的异常处理方式，要求多个catch中的异常不能相同，并且若catch中的多个异常之间有子父类异常的关系，那么子类异常要求在上面的catch处理，父类异常在下面的catch处理。

例如：

```
public static void test3() {
    int[] arr = {11, 22, 66, 0};
    try {
        System.out.println(arr[5]);
        System.out.println(arr[0] / arr[arr.length-1]);
    } catch (Exception e) {
        e.printStackTrace();
    } catch (ArithmeticException e) {
        System.out.println("除数");
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("数组下标越界");
    }
}
```

已捕获异常，为什么呢，因为Exception是异常的父类，既然父类已经捕获异常了，为啥还要用子类的呢？所以要删除

Exception 'java.lang.ArithmeticException' has already been caught

Delete catch for 'java.lang.ArithmeticException' Alt+Shift+Enter More actions... Alt+Enter

知乎 @敲代码的程序员

try-finally 形式：

```
try{
    //（尝试运行的）程序代码
}finally{
    //异常发生，总是要执行的代码
}
```

try-finally表示对一段代码不管执行情况如何，都会走 finally 中的代码，

例如：

```
public class TryFinallyDemo {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;
        try{
            System.out.println(a / b);
            System.out.println("会走try吗");
        }finally{
            System.out.println("会finally吗");//会finally吗
        }

        System.out.println("会走外面吗");
        /*
            没有捕获的话，他只会走finally语句然后报出异常。
            会finally吗
            Exception in thread "main" java.lang.ArithmeticException: / by ze
                at com.it.test3.TryFinallyDemo.main(TryFinallyDemo.java:8)
        */
    }
}
```

可以看到程序异常了，还是会去走finally语句块的代码。

try-catch-finally 形式：

```
try {
    // 可能会发生异常的程序代码
} catch (异常类型A e){
    // 捕获并处置try抛出的异常类型A
} finally {
    // 无论是否发生异常，都将执行的语句块
}
```

跟try-finally一样表示对一段代码不管执行情况如何，都会走 finally 中的代码。

当方法中发生异常，异常处之后的代码不会再执行，如果之前获取了一些本地资源需要释放，则需要在方法正常结束时和 catch 语句中都调用释放本地资源的代码，显得代码比较繁琐，finally 语句可以解决这个问题。

例如：

```
public class TryCatchFinallyDemo {  
    public static void main(String[] args) {  
        test();  
    }  
  
    public static void test() {  
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd"  
        Date date = null;  
        try {  
            //date = simpleDateFormat.parse("2020-10-06");//第一次运行成功  
            date = simpleDateFormat.parse("2020年10月06日");  
        } catch (ParseException e) {  
            e.printStackTrace();  
        }finally{  
            System.out.println("finally这里一定会执行");  
        }  
  
        System.out.println("会走外面这里吗" + date);  
    }  
}
```

运行成功的代码后结果：

```
finally这里一定会执行  
会走外面这里吗Tue Oct 06 00:00:00 CST 2020
```

运行失败的代码后结果：

```
java.text.ParseException: Unparseable date: "2020/10/06"
    at java.text.DateFormat.parse(DateFormat.java:366)
    at com.it.test3.TryCatchFinallyDemo.test(TryCatchFinallyDemo.java:19)
    at com.it.test3.TryCatchFinallyDemo.main(TryCatchFinallyDemo.java:12)
```

finally这里一定会执行
会走外面这里吗null

可以看到，无论失败，都会执行finally语句块的代码。


- 注意：try-catch-finally中，如果catch中 return了，finally还会执行吗？

```
public class TryCatchFinallyDemo2 {
    public static void main(String[] args) {
        test();
    }
    public static void test() {
        int a = 10;
        try{
            System.out.println(a / 0);
        }catch(ArithmeticException e) {
            e.printStackTrace();
            return ;
        }finally {
            System.out.println("finally");
        }
    }
}
```

运行结果：

```
java.lang.ArithmeticException: / by zero
```

```
at com.it.test3.TryCatchFinallyDemo2.test(TryCatchFinallyDemo2.java:1
at com.it.test3.TryCatchFinallyDemo2.main(TryCatchFinallyDemo2.java:5
finally
```



可以看到，就算catch中 return了，finally也会执行。

那finally是在return前呢，还是return后呢？

让我们看下面的代码？

```
public class TryCatchFinallyDemo2 {
    public static void main(String[] args) {
//        test();
        System.out.println(test2()); // 我有执行到吗 try
        System.out.println(test3()); // 我有执行到吗 catch
    }
    public static String test3() {
        String str = "";
        try {
            str = "try";
            System.out.println(10 / 0);
            return str;
        } catch (Exception e) {
            str = "catch";
            return str;
        } finally {
            str = "finally";
            System.out.println("我有执行到吗");
        }
    }
    public static String test2() {
        String str = "";
        try {
            str = "try";
            return str;
        } catch (Exception e) {
```

```
        str = "catch";
        return str;
    }finally {
        str = "finally";
        System.out.println("我有执行到吗");
    }
}
```

运行结果：

看到这里发现无论是否异常，finally都会执行，但是都在在return之前就执行了代码。可是为什么返回出来的字符串不是finally呢？让我们一起来思考思考：

- 我们看test2()方法，程序执行try语句块，把变量str赋值为"try"，由于没有发现异常，接下来执行finally语句块，把变量str赋值为"finally"，然后return str，则t的值是finally，最后str的值就是"finally"，程序结果应该显示finally，但是实际结果为“try”。
- 实际上，在try语句的return块中，当我们执行到return str这一步的时候呢，这里不是return str 而是return “try”了，这个放回路径就已经形成了。相当于return返回的引用变量（str是引用类型）并不是try语句外定义的引用变量str，而是系统重新定义了一个局部引用str2，返回指向了引用str2 对应的值，也就是"try"字符串。但是到这里呢，它发现后面还有finally，所以继续执行finally的内容，str = "finally"; System.out.println("我有执行到吗");，再次回到以前的路径,继续走return “try”，形成返回路径之后，这里的return的返回引用就不是变量str了，而是str2引用的值"try"字符串。

是不是对这个现象有了一定的了解。这里我们再转换下：

```
public class TryCatchFinallyDemo2 {
    public static void main(String[] args) {
        //      test();
    }
}
```

```
//      System.out.println(test2()); // try
//      System.out.println(test3()); // catch
      System.out.println(test4());
  }
  public static String test4() {
      String str = "";
      try {
          str = "try";
          return str;
      } catch (Exception e) {
          str = "catch";
          return str;
      } finally {
          str = "finally";
          return str;
      }
  }
}
```

这里我们猜测下，结果是什么呢？

运行结果：finally

- 我们发现try语句中的return语句给忽略。可能JVM认为一个方法里面有两个return语句并没有太大的意义，所以try中的return语句给忽略了，直接起作用的是最后finally中的return语句，就又重新形成了一条返回路径，所以这次返回的是“finally”字符串。

再看一下： 我们是不是知道finally语句是一定执行的，但是能有办法使他不执行吗？

```
public class TryCatchFinallyDemo3 {
    public static void main(String[] args) {
        try{
            System.out.println(10 / 0);
        }
```

```
        }catch(Exception e) {  
            e.printStackTrace();  
            System.exit(0);  
        }finally {  
            System.out.println("finally我有执行到吗");  
        }  
    }  
}
```

执行结果：

```
java.lang.ArithmeticException: / by zero  
    at com.it.test3.TryCatchFinallyDemo3.main(TryCatchFinallyDemo3.java:6
```

可以发现：

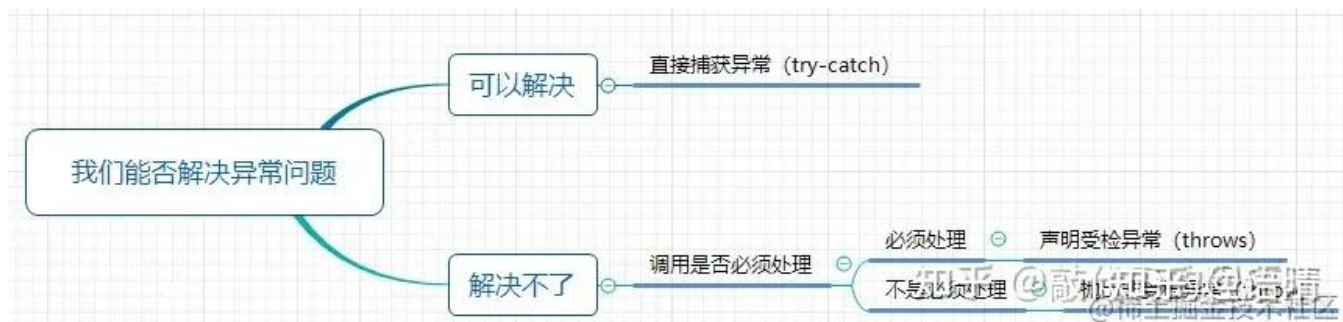
当只有在try或者catch中调用退出JVM的相关方法,此时finally才不会执行,否则finally永远会执行。

小结

1. try块：用于捕获异常。其后可接零个或多个catch块，如果没有catch块，则必须跟一个finally块。
2. catch块：用于处理try捕获到的异常。
3. finally块：无论是否捕获或处理异常，finally块里的语句都会被执行。当在try块或catch块中遇到return语句时，finally语句块将在方法返回之前被执行。在以下4种特殊情况下，finally块不会被执行：
4. 在finally语句块中发生了异常。
5. 在前面的代码中用了System.exit()退出程序。
6. 程序所在的线程死亡。
7. 关闭CPU。

如何选择异常类型

可以根据下图来选择是捕获异常，声明异常还是抛出异常



我们在日常处理异常的代码中，应该遵循的原则：

- 不要捕获类似Exception 之类的异常，而应该捕获类似特定的异常，方便排查问题，而且也能够让其他人接手你的代码时，会减少骂你的次数。
- 不要生吞异常。这是异常处理中要特别注重的事情。如果我们不把异常抛出来，或者也没有输出到日志中，程序可能会在后面以不可控的方式结束。有时候需要线上调试代码。

JDK1.7有关异常新特性

try-with-resources

Java 类库中有许多资源需要通过 close 方法进行关闭。比如 InputStream、OutputStream等。作为开发人员经常会忽略掉资源的关闭方法，导致内存泄漏。当然不是我啦！

在JDK1.7之前呢，try-catch-finally语句是确保资源会被关闭的最佳方法，就算异常或者返回也一样可以关闭资源。

- 让我们先看看之前我们如何关闭资源吧：

```
public static String readFile(String path) {  
    BufferedReader br = null;  
    try {
```

```
        br = new BufferedReader(new FileReader(path));
        return br.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    } finally { //必须在这里关闭资源
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

是不是我们必须finally语句块中手动关闭资源，否则会导致资源的泄露

- 在JDK1.7及以后的版本：JDK1.7 将其引入了try-with-resources 语句。
- try-with-resources 语句是一个声明一个或多个资源的try语句。try-with-resources 语句确保在语句的最后每个资源都被关闭，只要是实现了AutoCloseable接口或者是Closeable接口的对象都可以使用try-with-resources 来实现异常处理和关闭资源。
- 实际上，在编译时也会进行转化为try-catch-finally语句。

那我们来看看怎么使用吧：

格式：

```
try (创建流对象语句, 如果多个,使用';'隔开) {
    // 读写数据
} catch (IOException e) {
    e.printStackTrace();
}
```


演示下：

```
/**
 * JDK1.7之后就可以使用try-with-resources,不需要
 * 我们在finally块中手动关闭资源
 */
public class TryWithResourcesDemo {
    public static String readLineFormFile(String path) {
        try (BufferedReader br = new BufferedReader(new FileReader(path))) {
            return br.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

- 两者的对比：
- 代码精炼，在JDK1.7之前都有finally块，如果使用一些框架可能会将finally块交由框架处理，如Spring。JDK1.7及以后的版本只要资源类实现了AutoCloseable或Closeable程序在执行完try块后会自动close()所使用的资源无论br.readLine()是否抛出异常。
- 代码更完全。在出现资源泄漏的程序中，很多情况是开发人员没有或者开发人员没有正确的关闭资源所导致的。JDK1.7之后采用try-with-resources 的方式，则可以将资源关闭这种与业务实现没有很大直接关系的工作交给JVM 完成。省去了部分开发中可能出现的代码风险。
- 以readLineFormFile方法为例，如果调用 readLine()和 close()方法都抛出异常，后一个异常就会被禁止，以保留第一个异常。

catch多种异常并抛出新的异常

- 在JDK1.7之前catch 多个异常是这样的：

```
try{  
    编写可能会出现异常的代码  
}catch(异常类型A e){ 当try中出现A类型异常,就用该catch来捕获.  
    处理异常的代码  
}catch(异常类型B e){ 当try中出现B类型异常,就用该catch来捕获.  
    处理异常的代码  
}
```

- JDK1.7及以后可以这样：

```
try{  
    编写可能会出现异常的代码  
}catch (异常类型A | 异常类型B e) {  
    处理异常的代码  
}
```

但是呢，这个是同类型异常才可以这样定义。

自定义异常

为什么需要自定义异常类

我们说了Java中不同的异常类,分别表示着某一种具体的异常情况,那么在开发中，当Java内置的异常都不能明确的说明异常情况的时候，需要创建自己的异常。例如年龄负数问题,考试成绩负数问题。

什么是自定义异常类呢

在开发中根据自己业务的异常情况来自己定义异常类。例如：登录系统中，年龄能为负数吗，不能就需要自己定义一个登录异常类。

怎样自定义异常类

1. 自定义一个编译期异常: 自定义类并继承于`java.lang.Exception`。
2. 自定义一个运行时期的异常类:自定义类并继承于`java.lang.RuntimeException`。

一般定义一个异常类需要包含两个构造函数: 一个无参构造函数和一个带有详细描述信息的构造函数

```
public class MyException extends Exception {  
    public MyException(){ }  
    public MyException(String message){  
        super(message);  
    }  
}
```

举例

需求: 我们模拟登陆操作, 如果用户名已存在, 则抛出异常并提示: 亲, 该用户名已经被注册。这个相信大家也经常见到吧。

1.首先定义一个登陆异常类LoginException :

```
/**  
 *      登陆异常类  
 */  
public class LoginException extends Exception {  
    public LoginException() {  
    }  
    public LoginException(String message) {  
        super(message);  
    }  
}
```

2.模拟登陆操作, 使用数组模拟数据库中存储的数据, 并提供当前注册账号是

否存在方法用于判断。

执行结果：注册成功。

```
public class LoginTest {
    // 模拟数据库中已存在账号
    private static String[] names = {"hello", "world", "fish"};

    public static void main(String[] args) {
        //调用方法
        try{
            // 可能出现异常的代码
            checkUsername("fish");
            System.out.println("注册成功");//如果没有异常就是注册成功
        } catch(LoginException e) {
            //处理异常
            e.printStackTrace();
        }
    }
    //判断当前注册账号是否存在
    //因为是编译期异常，又想调用者去处理 所以声明该异常
    public static boolean checkUsername(String uname) throws LoginException {
        for (String name : names) {
            if(name.equals(uname)){//如果名字在这里面 就抛出登陆异常
                throw new LoginException("亲"+name+"已经被注册了! ");
            }
        }
        return true;
    }
}
```

编译时异常

编译器错误消息在Java软件代码在编译器执行时产生。需要重点记住的是，一个编译器可能为一个错误抛出多个错误消息。所以修复第一个错误并重编译，